#12 Appeal Brief 12/24/03 DH

Docket No. AUS9-2000-0277-US1

PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: **Browning et al.**

Serial No. **09/620,714**

Filed: **July 20, 2000**

For: **Method and Apparatus to Debug a Program from a Predetermined Starting Point**

Group Art Unit: **2122**

Examiner: **Steelman, Mary J.**

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

ATTENTION: Board of Patent Appeals and Interferences

Certificate of Mailing Under 37 C.F.R. § 1.8(a)
I hereby certify this correspondence is being deposited with the United States Postal Service as First Class mail in an envelope addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on December 19, 2003.

By: Rebecca Clayton
Rebecca Clayton

12/30/2003 MDAMTE1 00000027 090447 09620714
01 FC:1402 330.00 DA

APPELLANT'S BRIEF (37 C.F.R. 1.192)

This brief is in furtherance of the Notice of Appeal, filed in this case on October 30, 2003.

The fees required under § 1.17(c), and any required petition for extension of time for filing this brief and fees therefore, are dealt with in the accompanying TRANSMITTAL OF APPEAL BRIEF.

This brief is transmitted in triplicate. (37 C.F.R. 1.192(a))

REAL PARTIES IN INTEREST

The real party in interest in this appeal is the following party: International Business Machines Corporation.

RELATED APPEALS AND INTERFERENCES

With respect to other appeals or interference's that will directly affect, or be directly affected by, or have a bearing on the Board's decision in the pending appeal, there are no such appeals or interference's.

STATUS OF CLAIMS

A. TOTAL NUMBER OF CLAIMS IN APPLICATION

Claims in the application are: 1-29

B. STATUS OF ALL THE CLAIMS IN APPLICATION

1. Claims canceled: None
2. Claims withdrawn from consideration but not canceled: None
3. Claims pending: 1-29
4. Claims allowed: None
5. Claims rejected: 1-29

C. CLAIMS ON APPEAL

The claims on appeal are: 1-29

STATUS OF AMENDMENTS

A response to final Office Action was filed by Appellants on September 3, 2003. The claims were not amended in such response.

SUMMARY OF INVENTION

A technique for debugging computer software. The claimed invention advances the state of the art by providing improved automation capabilities for the debugging of one or more processes in a multi-process environment. The improved debug automation advantageously

provides capabilities previously unattainable by providing the ability to automatically save *and* automatically restore a process state. Such automation provides significant improvements in debugging by allowing a process to run until some event occurs, saving the process state into a checkpoint file, modifying registers or other variables, and then resuming execution - all under program control. This is particularly useful for long running programs which may not crash until several hours or days of running. For example, per the claimed invention, it is possible to automatically resume execution from a saved process state in response to a predefined event, providing improvements in debug capabilities by allowing a programmer who is debugging a process to instruct the debugger to run between a checkpoint and breakpoint repeatedly using a plurality of different register or memory variable values in order to assist in problem determination.

In particular, the present invention provides a debugger combined with a child process that can be checkpointed and restarted. This may be done automatically under program control of the debugger. In a preferred embodiment, the debugger, executing in memory as a process, creates a child process for the program being debugged. The child process can then, in turn, create further child processes. The debugger, which is the parent process, has control over the child process(es). The debugger may save the image of all processes under control of the debugger in a checkpoint file to resume debugging from that state. Thus, the present invention provides a mechanism for automatically resuming debugging from a saved state, allowing the programmer to modify registers and memory variables and resume debugging from such known state (Specification page 9, lines 17-31). This use of a debugger to recreate processes from a checkpoint file can potentially save a user several hours or even days of debugging efforts to debug a long-running process that crashes after several hours or days of execution if a checkpoint file containing a process image is created on some interval-basis (Specification page 10, lines 19-30).

A key enabling feature of the present invention is the ability to resume processing from a known state (that was previously saved) under process control of a parent debug process. Since the debugger process creates or spawns the child process which is the subject of the debug session, the debugger process can be programmed to checkpoint and restore the child process using a multitude of different operating conditions such as variable or memory values (Specification page 11, 1-10), thus significantly advancing the state of the art.

ISSUES

I. 35 U.S.C. § 102, Anticipation

Whether the Examiner properly rejected Claims 1-6, 11-18 and 23-29 under 35 U.S.C. 102(e) as being anticipated by U.S. Patent 6,240,529 to Kato.

II. 35 U.S.C. § 103, Obviousness

A. Whether the Examiner properly rejected Claims 7-9 and 19-21 under 35 U.S.C. 103 as being unpatentable over U.S. Patent 6,240,529 to Kato, and further in view of U.S. Patent 5,560,009 to Lenkov et al.

B. Whether the Examiner properly rejected Claims 10 and 22 under 35 U.S.C. 103 as being unpatentable over U.S. Patent 6,240,529 to Kato, and further in view of U.S. Patent 6,412,106 to Leask et al.

GROUPING OF CLAIMS

The claims do not stand or fall together, and Appellants consider the following grouping of claims to be separately patentable:

Group I: Claims 1-5, 7-17 and 19-25

Group II: Claims 6 and 18

Group III: Claims 26 and 27

Group IV: Claim 28

Group V: Claim 29

The claims of Group II are separately patentable by reciting the claimed feature of reinitiating debugging from the stored process state, wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process.

The claims of Group III are separately patentable by reciting the claimed feature that the predefined event (to which the retrieval of the stored process state of Claim 1 is responsive to) is a checkpoint.

The claims of Group IV are separately patentable by reciting the claimed features of creating a child process from the debug process, saving a process state of the child process (the child process having been created from a debug process) and executing the child process using the stored process state.

The claims of Group V are separately patentable by reciting the claimed feature of saving a process state of a traced process and a process state of another process that is not being traced by the debugger.

ARGUMENT

I. 35 U.S.C. § 102, Anticipation

The Examiner rejected Claims 1-6, 11-18 and 23-29 under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent 6,240,529 to Kato.

GROUP I

With respect to Claim 1, such claim recites the claimed step of "retrieving the stored process state *in response to a predefined event*". This predefined event is a key enabling feature of the present invention, allowing for retrieval of a previously saved process state in response to this predefined event to thereby allow subsequent execution of the process from a known state. In rejecting Claim 1, the Examiner cites Kato Col. 8, lines 29-35 as reading on this claimed step. Appellants show that to the contrary, this passage states:

> "*When a state restoration command is issued*, storage situation management unit 121 displays a list of currently existing state storage files 118 based on storage situation management file 122. FIG. 4 shows an example of the list. *If the file to be restored is selected from the information*, then state storage restoration unit 117 reads in the file and restores the debugged state."

It can clearly be seen that Kato's restoration is responsive to a state restoration command *currently* being issued. This does not teach or otherwise disclose that claimed step of retrieving the stored process state *in response to a predefined event*. The cited reference requires a *manual restoration* of a state storage file as selected by a user from a list of existing files (Kato Col. 4, lines 20-24; Col. 8, lines 29-35; Figure 4; Col. 10, lines 3-10; Col. 11, lines 17-29). Kato discusses problems with retrieving a state storage file at Col. 4, line 62 – Col. 5, line 11, but this retrieval is *responsive to a user input command* (Kato Col. 4, lines 20-24).

It should particularly be noted that the primary purpose of the Kato teachings is to assist the user in selecting which file from a plurality of saved state storage files to choose, as described by Kato at Col. 11, lines 17-28 and reproduced as follows:

While the conventional trace information is such as described above, in Embodiment 2, the debugged state storage file name is recorded into 407 at a timing at which the debugged state is stored. Consequently, while, in Embodiment 1, the retrieval means for a state storage file only refers to the state storage file list of FIG. 4, in Example 2, also the retrieval means which refers to trace information is additionally provided, and **specification of a debugged information file to be restored is facilitated by confirming a state storage file, a type of an execution instruction upon production of the file, a state of a memory and so forth.**

This is accomplished by saving state storage files names in a trace buffer so that the user can look at the trace buffer (see Kato Figure 6) to determine which stage storage file had just been saved prior to a system crash, as described by Kato at Col. 10, line 47 – Col. 11, line 16 and reproduced as follows:

"More particularly, upon storage of a debugged state into a state storage file and registration into a storage situation management file in step 313 of FIG. 5A, information of the state storage file name and so forth is stored also into a buffer for trace information storage. Although the timing for storage of trace information appears also when a state storage command is issued or when a debugging ending command is issued (330, 336), if trace information can be acquired also at such timings, then registration into the trace buffer is performed simultaneously. An example of trace information in which debugged state storage file names are stored in this manner is shown in FIG. 6.

FIG. 6 is a view showing an example wherein state storage file names are filled into trace information by the storage situation management unit of Embodiment 2. In the example shown, two state storage information file names are filled in the trace information.

Here, the trace information is described. The trace frame item of 401 indicates a number of a trace frame. An older number indicates trace information acquired at an earlier timing. The time information of 402 indicates a required

time from starting of execution of contents of the just preceding trace to starting of execution of contents of the current trace, as represented in units of a clock. 403 indicates a trace of a result of fetching of an instruction of a program. M1 represents fetching of the first byte of the instruction. OP represents fetching of the other bytes of the instruction. 404 indicates a result of accessing to data upon execution of the program. RP represents data reading by the program, and WP represents data writing by the program. 405 indicates an item which indicates an address of a branching destination. 406 indicates a disassemble representation of an instruction. **From a disassemble representation, it can be discriminated to which instruction the trace information corresponds.**" (emphasis added by Appellants

The key point being that the teachings of the Kato reference are directed to *assisting the user in manually selecting* from one of many state storage files to be used when *the user manually issues a state restore command* (see, for example, Kato Figure 6, element 407). The teaching of a user entering a manual state restore command – albeit under assistance of state storage file names being included in a trace buffer to assist in such manual selection – cannot reasonably be construed to read on the claimed feature of retrieving the stored process state *in response to a predefined event*. The claimed feature of retrieving the stored process state in response to a predefined event advantageously allows for a process to automatically run repeatedly between a checkpoint and breakpoint using a plurality of register or memory variable values to thereby improve debugging capabilities (Specification page 16, line 28 – page 17, line 3). As the cited reference does not teach such retrieval in response to a *predefined* event, but rather responsive to a *current* input command (Kato Figure 5A, step 302 and Figure 5B, step 331), the cited reference does not anticipate Claim 1.

Nor would it be obvious to modify Kato to include such a retrieval of state information in response to a predefined event. Kato teaches that state information can be *stored* when a certain event occurs (Col. 9, lines 64-67), but requires manual user intervention in the *restore* of such state information (Col. 10, lines 3-10; Col. 11, lines 17-29). Since Kato was aware of a desire to automatically store state information, but yet provided no solution for retrieval of state

information in response to a predefined event, it must not have been obvious to Kato as to how to accomplish this. In addition, Kato's express purpose to improve retrieval of state information is to simultaneously store situation information in a storage situation management file when state information is stored, to facilitate a *manual determination* of which among a plurality of existing state storage files should be manually selected by the user (Col. 5, lines 35-48) based upon the user's manual input of a restore command at the time the restore command is desired (Col. 10, lines 3-10; Figure 5B, element 322). Modifying Kato in accordance with the claimed invention would defeat this expressed purpose.

GROUP II

With respect to Claim 6, the cited reference does not teach the claimed step of "reinitiating debugging from the stored process state, wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process". Notably absent from the Examiner's reasoning in rejecting Claim 6 is any mention or discussion of this claimed step. The Examiner merely asserts that the cited reference teaches "reinitiating debugging from the stored process state". Appellants show that Claim 6 goes beyond this assertion, and specifically recites "wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process" in addition to the alleged "reinitiating debugging from the stored process state". Thus, there are at least two processes recited in Claim 6: (1) a process, and (2) at least one child process, where the process has control over the at least one child process. For a prior art reference to anticipate in terms of 35 U.S.C. 102, every element of the claimed invention must be identically shown in a single reference. *In re Bond*, 910 F.2d 831, 15 USPQ2d 1566 (Fed. Cir. 1990). As every element of the claimed invention is not identically shown in a single reference – and in fact has not even been alleged to be shown in a single reference – Claim 6 is shown to have been erroneously rejected under 35 U.S.C. § 102 as being anticipated by U.S. Patent 6,240,529 to Kato, as there is no teaching of "wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process".

This invention of Claim 6 advantageously enables debugging in a multi-processing environment. In a traditional multi-processing environment, a process typically creates or

spawns child processes which run concurrently with the parent process (Specification page 9, lines 7-16). In order to provide debugging capabilities in such a multi-processing environment, per the claimed invention, it is critical to keep track of these child processes *in addition to* the process being debugged. Inclusion of process descriptors for each child process in the process state of the process being debugged provides this capability. This advantageously provides process state information for both (1) the process and (2) the at least one child process. The cited reference only alludes to maintaining debug information for a single running process. Thus, the invention of Claim 6 provides a significant advancement in the art by providing debug capabilities in an environment with multiple concurrently running processes.

GROUP III

Claim 26 depends upon Claim 1 and is shown to not be anticipated by the cited reference for reasons given above regarding Claim 1.

Further with respect to Claim 26 (and dependent Claim 27), such claim recites that the predefined event (to which the retrieval of the stored process state of Claim 1 is responsive to) is a checkpoint. This checkpoint is shown in the preferred embodiment by restore checkpoint element 616 in Figure 6. As described at Specification page 16, line 28 – page 17, line 3, by providing the ability to restore an image of the process being debugged in response to an event – and in this case a restore checkpoint – the debugger may advantageously be instructed to automatically run between a checkpoint and a breakpoint repeatedly. This claimed feature advantageously provides the ability to repeatedly run between the checkpoint and breakpoint using a plurality of register or memory variable values to assist in debugging the process.

The cited reference does not teach the claimed steps of "retrieving the stored process state in response to a predefined event" ... "wherein the predefined event is a checkpoint". As can clearly be seen by this claim language, the step of retrieving the stored process state is *in response to a checkpoint*. The Examiner cites Kato Fig. 5A, #309 and #314 and Col. 9, lines 15-36 as reading on this claimed step. Appellants show that to the contrary, step #309 merely checks whether an event has occurred and if so, processing of the action registered in connection with the event is performed (Kato Col. 9, lines 15-21). There is no mention whatsoever of any type of specific action of *retrieval of a stored process state in response to this event checking.*

Nor does Kato's teaching with regard to step #314 overcome this deficiency. Step #314 determines if a break occurred. If YES, the execution flag is changed to OFF and execution processing ends (Col. 9, lines 35-38). If NO, a determination is made on *whether to continue execution of instructions* (Col. 9, lines 38-43). Again, this has absolutely nothing whatsoever to do with retrieval of the stored process state *in response to a checkpoint*, as claimed in Claim 26. Therefore, Claim 26 (and dependent Claim 27) is shown to not be anticipated by the cited reference as there are missing claimed elements.

Further with respect to Claim 26 (and dependent Claim 27), Appellants show that the cited reference does not teach the claimed step of "repeatedly running between the checkpoint and the breakpoint for a plurality of times". The Examiner cites this same passage at Kato Col. 9, lines 15-36 as teaching this claimed feature. This passage is repeated below in its entirety to clearly show there is no mention whatsoever of repeatedly running between the checkpoint and the breakpoint for a plurality of times:

> "Then, it is checked whether or not a state corresponding to one of events registered by the user is satisfied as a result of the execution of the instruction (309), and if an event is detected (310), then processing of an action registered in connection with the event is performed (311).
>
> By the way, in the present embodiment, a function to storage a debugged state into a file if a certain event occurs is provided in addition to the conventional functions. Thus, it is assumed that an event for which storage processing of a debugged state is to be performed when the event occurs is registered in advance. In this instance, if the event is detected, then a state storage request is generated (312), and a current debugged state is stored into state storage file 118.
>
> In the present embodiment, since also the function to manage a situation when a debugged state is stored is additionally provided, the situation upon the storage is registered simultaneously into storage situation management file 122 (313). If a request for a break is generated in response to the detection of the event (314), then the execution flag is changed to OFF (316), thereby ending the execution processing."

Appellants show that this cited passage has nothing to do with "repeatedly running between the checkpoint and the breakpoint for a plurality of times". Rather, it teaches that if an event is detected, processing of an action registered in connection with the event is performed (Col. 9, lines15-20). This is a general statement that makes no mention of what the registered action is or does. It merely states that the registered action 'is performed'. The cited passage goes on to state that a state storage request is generated and the current debugged state is stored into state storage file 118 (Col. 9, lines 27-29) along with the situation (Col. 9, lines 30-35). This is not seen to teach or otherwise disclose in any fashion the claimed step of "repeatedly running between the checkpoint and the breakpoint for a plurality of times". Rather, Kato merely goes back to its initial state awaiting a next user input command (Kato Figure 5A, steps 301 and 302). Hence, Claim 26 (and dependent Claim 27) is further shown to not be anticipated by the cited reference.

GROUP IV

Claim 28 advantageously provides a method for debugging a process in a multi-process environment. This is accomplished by initiating a debug process and creating a child process from the debug process. As described above in the Summary of the Invention section, creation of the child process from the debug process advantageously allows the debug process to control the child process (which is the primary focus of the debug session due to the recitation in the claim of saving a process state of the child process, retrieving the stored process state, and executing the child process using the stored process state) and thus advantageously provides an ability to completely automate checkpoint and restore operations for the child process. In one example of such automation enablement, it is possible to program the debug process to perform multiple checkpoint and restore operations of the process it is controlling (the child process) while varying various other system parameters such as registers and memory to thereby facilitate a more robust debug session.

With respect to Claim 28, Appellants show that the cited reference does not teach the claimed steps of "creating a child process from the debug process", "saving a process state of the child process" (the child process having been created from a debug process) and "executing the child process using the stored process state". In rejecting Claim 28, the Examiner states that Kato Col. 7, line 15 reads on the claimed step of initiating a debug process, and that Kato Col.

10, lines 43-46 reads on the claimed step of creating a child process from the debug process. Appellants show error in this assertion as follows.

Claim 28 recites two processes – a debug process and a child process (which is created from the debug process). Col. 7, line 15 of Kato (which the Examiner asserts reads on the claimed debug process) reads as follows:

> "When debugging is started a down load command is inputted from information IN/Out apparatus 102 to down load object code file 104 of a program *which is an object of debugging to the debugging apparatus*." (emphasis added by Appellants)

So, according to this reading of Kato, the object code file 104 is the object of debugging. Of particular importance is that according to Claim 28, such claim recites that the *child process* (created from the debug process) is the object of debugging, as it goes on to state "saving a process state of the child process", retrieving the stored process state" and executing the child process using the stored process state". Per the Examiner's interpretation of Kato, the object code file 104 is the debug process that is the object of debugging (as the Examiner states that Kato Col. 7, line 15 reads on the claimed debug process). This is not merely a play on words, but a very real difference in that the claim recites that the process being debugged (the child process) is created from another process (the debug process). This claimed feature advantageously allows the debug process to control and thus automate debugging of the child process. As the passage cited by the Examiner states it is the object code file which is the object of debugging, which is contrary to what is claimed (the child process created from the debug process is the object of the debugging), it is shown that Claim 28 is not anticipated by this reading of Kato.

Nor does the Kato passage cited at Col. 10, lines 43-46 overcome this deficiency. There it states (the entire sentence is being reproduced herewith):

> "When setting so that trace information is also acquired simultaneously at a timing of storage of a debugged state is performed in this manner, in Embodiment 2, the storage situation management unit in Embodiment 1 additionally has a function of recording a debugged state storage timing into trace information."

This passage states that debugged state storage timing is stored into trace information. This merely states that two things are being stored, not that two processes are running, where one process (the child process) is created by the other and this one process (child process) is the process being debugged (by saving and retrieving its associated state information). Restated, a teaching of storing two things in a file does not teach or otherwise disclose a debug process that creates a child process, where the child process is the object of debugging (by saving and retrieving the child's process state for use during subsequent execution of the child process).

The Examiner goes on to discuss various aspects of Kato as reading on the claimed saving, retrieving, and executing steps. These are all with respect to the object code file 104 which is the object of debugging (Kato Col. 7, lines 11-20), which the Examiner has equated with the claimed debug process. As stated above, the debug process and the child process are very different, with a unique relationship where the child process (the subject of the claimed steps of saving, retrieving, and executing) is created from the debug process. Thus, it is shown that saving, retrieving and executing of an object code file is very different from the claimed steps of saving, retrieving and executing from a child process created from the debug process. Therefore, Claim 28 is further shown to not be anticipated by the cited reference.

GROUP V

Claim 29 recites saving two different process states – the process state of a traced process and the process state of another process that is not being traced. The claimed feature advantageously allows checkpointing processes and saving their process images in a checkpoint file even when they are not being traced (Specification page 10, lines 19-21).

In particular, Claim 29 recites "saving a process state of a traced process and a process state of another process that is not being traced by the debugger". The cited reference does not teach or otherwise disclose saving a process state of a traced process *and* a process state of another process that is not being traced. The Examiner cites Kato Col. 7, lines 35-47 and 65-66 as reading on this claimed step. These passages are reproduced below:

> "Debugger unit 106 in debugging apparatus 103 performs processing regarding a debugging function for breaking, tracing or the like. A break point at which execution of a program is interrupted when a certain event occurs and a trace

wherein, when a certain event occurs, an instruction execution situation, a memory access situation and so forth at the point of time are recorded and individually set with commands in advance. The events set with the commands and actions such as breaking and tracing when such events occur are registered in event processing unit 110 and action processing unit 111 in debugger unit 106 and are processed in the following manner when the instructions are executed." (Col. 7, lines 35-47)

"In the debugging apparatus of the present embodiment, event processing unit 110 has, in addition to the conventional functions, a function of storing a debugged state into a file if a certain event occurs." (the entire sentence encompassing Col. 7, lines 65-66)

As can be seen, the passage at Col. 7, lines 35-47 generally discusses a debugging apparatus that has the ability to set a breakpoint and perform tracing when a certain event occurs, such as an instruction execution situation or a memory access situation. These are well known debugging operations and are not seen to be of importance with regards to Claim 29 and specifically the saving of process states from two different processes. The passage cited at Col. 7, lines 65-66 merely describes an ability to store a debugged state into a file if a certain event occurs. Thus, it is shown that this passage merely recites storing a debugged state if a certain event occurs. It does not teach or otherwise disclosure saving a process state *of the traced process* and a process state of *another process that is not being traced by the debugger*. At best, it teaches storing a single process state, whereas the claim is directed to storing two different process states from two different processes – with one state being traced and the other one not being traced. Thus, it is shown that Claim 29 is not anticipated by the cited reference.

II. 35 U.S.C. § 103, Obviousness

A. The Examiner rejected Claims 7-9 and 19-21 under 35 U.S.C. § 103 as being unpatentable over U.S. Patent 6,240,529 to Kato, and further in view of U.S. Patent 5,560,009 to Lenkov et al. To establish prima facie obviousness of a claimed invention, all of the claim

limitations must be taught or suggested by the prior art. MPEP 2143.03. See also, *In re Royka*, 490 F.2d 580 (C.C.P.A. 1974). In the absence of a proper prima facie case of obviousness, an applicant who complies with the other statutory requirements is entitled to a patent. See *In re Oetiker*, 977 F.2d 1443, 1445, 24 USPQ2d 1443, 1444 (Fed. Cir. 1992). Appellants show error in the rejection of Claims 7-9 and 19-21 for similar reasons to those given above regarding Claim 1, of which these claims ultimately depend upon. As there is at least one missing claimed element with regarding Claim 1, the Examiner has failed to establish that *all* the claimed limitations of these Claims 7-9 and 19-21 are taught or suggested by the cited references.

B. The Examiner rejected Claims 10 and 22 under 35 U.S.C. § 103 as being unpatentable over U.S. Patent 6,240,529 to Kato, and further in view of U.S. Patent 6,412,106 to Leask et al. Appellants show error in the rejection of Claims 10 and 22 for similar reasons to those given above regarding Claim 1, of which these claims ultimately depend upon. As there is at least one missing claimed element with regarding Claim 1, the Examiner has failed to establish that *all* the claimed limitations of these Claims 10 and 22 are taught or suggested by the cited references (per MPEP 2143.03; *In re Royka*, supra).

In conclusion, Appellants have shown that Claims 1-29 have been erroneously rejected by the Examiner, and request that the rejection of such claims be reversed by the Board of Appeals.

Duke W. Yee
Reg. No. 34,285

Wayne P. Bailey
Reg. No. 34,289
Carstens, Yee & Cahoon, LLP
PO Box 802334
Dallas, TX 75380
(972) 367-2001

APPENDIX OF CLAIMS

The text of the claims involved in the appeal are:

1. A method in a data processing system for debugging a process from a starting point, comprising:

initiating debugging of the process;

saving a process state in response to a first event to form a stored process state;

retrieving the stored process state in response to a predefined event; and

reinitiating debugging from the stored process state.

2. The method of claim 1, wherein the first event occurs periodically.

3. The method of claim 1, wherein the process state is saved in a checkpoint data structure.

4. The method of claim 3, wherein the checkpoint data structure is a checkpoint file.

5. The method of claim 3, wherein the checkpoint data structure includes a process descriptor for the process.

6. A method in a data processing system for debugging a process from a starting point, comprising:

initiating debugging of the process;

saving a process state in response to a first event to form a stored process state;

retrieving the stored process state in response to a second event; and

reinitiating debugging from the stored process state, wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process.

7. The method of claim 5, wherein the checkpoint data structure further includes at least one data type descriptor for at least one data type corresponding to the process.

8. The method of claim 7, wherein the checkpoint data structure further includes at least one instance descriptor for at least one instance of data corresponding to each of the at least one data type.

9. The method of claim 8, wherein the checkpoint data structure further includes at least one data block corresponding to each of the at least one instance descriptor.

10. The method of claim 1, further comprising the step of modifying at least one register or memory variable before resuming debugging from the stored process state.

11. The method of claim 1, wherein the process state is saved when the program is in a stopped state.

12. The method of claim 11, wherein the stopped state is at a breakpoint.

13. An apparatus for debugging a process from a starting point, comprising:

a processor; and

a memory electrically connected to the processor, the memory having stored therein a program to be executed on the processor for performing the following steps:

initiating debugging of the process;

saving a process state in response to a first event to form a saved process state;

retrieving the saved process state in response to a predefined event; and

reinitiating debugging from the saved process state.

14. The apparatus of claim 13, wherein the first event occurs periodically.

15. The apparatus of claim 13, wherein the process state is saved in a checkpoint data structure.

16. The apparatus of claim 15, wherein the checkpoint data structure is a checkpoint file.

17. The apparatus of claim 15, wherein the checkpoint data structure includes a process descriptor for the process.

18. An apparatus for debugging a process from a starting point, comprising:

a processor; and

a memory electrically connected to the processor, the memory having stored therein a program to be executed on the processor for performing the following steps:

initiating debugging of a process;

saving a process state in response to a first event to form a saved process state;

retrieving the saved process state in response to a second event; and

reinitiating debugging from the saved process state, wherein the process has control over at least one child process and the process state includes a process descriptor for each of the at least one child process.

19. The apparatus of claim 17, wherein the checkpoint data structure further includes at least one data type descriptor for at least one data type corresponding to the process.

20. The apparatus of claim 19, wherein the checkpoint data structure further includes at least one instance descriptor for at least one instance of data corresponding to each of the at least one data type.

21. The apparatus of claim 20, wherein the checkpoint data structure further includes at least one data block corresponding to each of the at least one instance descriptor.

22. The apparatus of claim 13, wherein program further comprises the step of modifying at least one register or memory variable before resuming debugging from the retrieved process state.

23. The apparatus of claim 13, wherein the process state is saved when the program is in a stopped state.

24. The apparatus of claim 23, wherein the stopped state is at a breakpoint.

25. A computer program product embodied in a computer readable medium for debugging a process from a starting point, comprising:

instructions for initiating debugging of the process;

instructions for saving a process state in response to a first event to form a saved process state;

instructions for retrieving the saved process state in response to a predefined event; and

instructions for reinitiating debugging from the saved process state.

26. The method of Claim 1, wherein the first event is a breakpoint and the predefined event is a checkpoint, and further comprising the step of repeatedly running between the checkpoint and the breakpoint for a plurality of times.

27. The method of Claim 26 wherein variable values are automatically modified after retrieving the stored process state for the checkpoint.

28. A method in a data processing system for debugging a process in a multi-process environment, comprising the steps of:

initiating a debug process;

creating a child process from the debug process;

saving a process state of the child process in response to a first event, to form a stored process state;

retrieving the stored process state in response to a second event; and

executing the child process using the stored process state.

29. A method in a data processing system for debugging a process in a multi-process environment, comprising the steps of:

tracing a process by a debugger;

saving a process state of the traced process and a process state of another process that is not being traced by the debugger;

retrieving the saved process states; and

reinitiating debugging of the process using the retrieved process states.